C++

Rahul S. Bansod

- → superset of C
- → object oriented
  - → data Abstraction
  - → encapsulation
  - → Inheritance
  - → polymorphism
- → secure & portable
- → STL
- → extensible
- → concept of pointer
- → fast
- → compiled language, oops, structural, functional...

The four pillars of object-oriented programming (OOPS) in C++ are

1) Encapsulation: This is the concept of building the data (variable) and methods (functions) that ~~people~~ operate on the data into a single unit called a class. It also involves restricting direct access to some of the object's components, which is done using specifiers like private, protected, and public.

2) Abstraction :- Abstraction involves hiding complex implementation details and showing only the essential features of an object. This helps in reducing complexity and allows the programmer to focus on interacting with the object's interface rather than its internal working.

3) Inheritance: Inheritance allows ~~one interface~~ a class to inherit properties & behaviors (methods) from another class. The class inherits is called the derived class, and the class being inherited from is called the base class. This promotes code reusability & logical hierarchy.

4) Polymorphism: Polymorphism allows one interface to be used for a general class of actions. The most common types are compile-time polymorphism (method overloading, operator overloading) and runtime polymorphism (method overriding). It enable objects of different classes to be treated as object of a common base class.

Encapsulation

→ class & object

access specifier → private, public, protected

Benefits of Encapsulation

→ Data hiding: Internal object details are hidden from the outside world. The user interact with the object only through public interfaces.

→ control: By providing getters & setters, you can impose rules on how data is modified (e.g:- validation)

→ modularity: The implementation details of the class can be changed without affecting other parts of the program that use it.

→ code maintainability: It makes the code more organized and easier to maintain.

e.g:-

```
#include <iostream>
using namespace std;

class Person {

private :
    string name;
    int age;
Public :
  void setName (string n) {
      name = n;
    }

  string getName () {
      return name;
    }
Void setAge (int a) {
    if (a >= 0) {
      age = a;
    }
  }
int getAge () {
    return age;
  }
};
```

```
int main()
  {
  Person person;
   person.setName("John");
   person.setAge(30);
  cout<<"Name : " << person.getName() << endl;
  cout <<"Age : " << person.getAge() << endl;

  return 0;
  }
```

e.g:- bank account

```
#include <iostream>
using namespace std;
class BankAccount {
private :
   double balance;

public:
BankAccount (double initialBalance) {
      if (initialBalance >0)
       {
        balance = initialBalance;
       } else {
        balance = 0;
        cout<<" Initial balance must be positive !" <<endl;
       }}
```

```
void deposit (double amount)
  {
    if (amount > 0)
      {
      balance += amount;
      cout << " Deposited :" << amount << endl;
      }
    else
      {
      cout << " Deposito amount must be positive!" << endl;
      }
  }
void withdraw (double amount)
  {
  if (amount > 0 && amount <= balance) {
      balance -= amount;
      cout << " withdraw : " << amount << endl;
      }
    else
      {
      cout << " Invalid withdrawal amount !" << endl;
      }
  }
double getBalance () const {
      return balance;
      }
};
```

```
int main()
{
BankAccount account(1000);
account.deposit(500);
account.withdraw(200);
cout<<"current Balance: "<< account.getBalance()<<
                                        endl;

return 0;
}
```

eg:- Rectangle

```
#include <iostream>
using namespace std;
class Rectangle {
private :
double length;
double width;

public :
Rectangle (double l, double w) {
    setLength(l);
    setwidth(w);
}
```

```
void setLength (double l)
{
  if (l>0)
  {
   length = l;
   }
   else
   {
   cout <<" Length must be positive!" << endl;
   }
}
void setWidth (double w)
{
  if(w>0)
  {
  width = w;
  }
  else {
  cout <<" width must be positive!" << endl;
  }
}
double getArea () const {
        return length * width;
  }
};
```

```
int main()
{
Rectangle rect(5,3);
cout<<" Area of rectangle: "<< rect.getArea()<< endl);
rect.setLength(6);
rect.setWidth(4);
cout<<"New area of rectangle: "<< rect.getArea()<<
                                    endl;

return 0;

}
```

e.g:→ Employee

```
#include <iostream>
using namespace std;
class Employee {
private:
string name;
double salary;
int employeeId;

Public:
Employee (string n, double s, int i.d)
{
setName(n);
setSalary(s);
setEmployeeId(id);
}
```

```
void setName (string n) {
    name = n;
}
void setSalary (double s) {
  if (s>0) {
    salary = s;
  }
  else
  {
  cout <<" salary must be positive !" << endl;
  }
}
void setEmployeeId (int id)
{
  if (id >0) {
    employeeId = id;
  }
  else
  {
  cout <<" Employee ID must be positive !" << endl;
  }
}
string getName() const {
    return name;
  }
double getSalary () const {
  return salary;
}
```

```
    int getEmployeeId() const {
        return employeeId;
    }
};
int main()
{
Employee emp("Alice", 5000, 101);
cout << " Employee ID : " << emp.getEmployeeId() << endl;
cout << " Employee Name : " << emp.getName() << endl;
cout << " Employee salary : " << emp.getsalary() << endl;

emp.setsalary(55000);
cout << "updated salary : " << emp.getsalary() << endl;

return 0;
}
```

e.g :- car

```
#include <iostream>
using namespace std;

class Car {
private:
 double speed;
 double fuel;
```

```
Public :
Car (double spd, double fl)
{
    setspeed (spd);
    set-Fuel (fl);
}
void setspeed (double spd)
{
    if( spd >=0)
    {
     speed =spd;
    }
    else
    {
    Cout <<" speed cannot be negative!" <<endl;
    }
}
void setfuel (double fl)
    {
     if (fl >=0 && fl <=100)
     {
       fuel = fl;
     }
     else
     {
      cout<<" Fuel level must be between 0 & 100!"<<en
     }
    }
```

```
    double .getSpeed() const {
        return speed;
    }
    double getFuel() const {
        return fuel;
    }
  void drive() {
    if (fuel > 0)
    {
  fuel -= speed * 0.1;
   if (fuel < 0)
   {
     fuel = 0;
   }
   cout << " Driving at speed " << speed << " with fuel level "
        << fuel << " % " << endl;
   }
   else {
    cout << " Not enough fuel ! " << endl;
   }
  }
};
int main() {
 Car myCar (60, 50);
cout << " speed : " << myCar.getSpeed() << " km/h " << endl;
cout << " Fuel : " << myCar.getFuel() << " % " << endl;
```

```
myCar.drive();
mycar.drive();

mycar.setspeed(80);
mycar.setfuel(40);
cout << "updated speed : " << mycar.getspeed() << " km/h" << endl;
cout << "updated fuel : " << mycar.getfuel() << " %" << endl;

mycar.drive();
return 0;
}
```

e.g:- Temperature converter (fahrenheit to kelvin)

```
#include <iostream>
using namespace std;
class Temperature {
private :
    double celsius;

public :
    Temperature(double temp)
    {
        setCelsius(temp);
    }
```

```
double getCelsius (double temp)
{
  celsius = temp;
  return celsius;
}
double tofahrenheit () const {
    return (celsius * 9/5) + 32;
}
double tokelvin () const {
    return celsius + 273.15;
}
};
int main ()
Temperature temp (25);
cout << " Temperature in celsius : " << temp.getCelsius ()
                                << " °C " << endl;
cout << "Temperature in fahrenheit : " << temp. tofahrenheit
                                << "°F" << endl;
cout << "Temperature in kelvin : " << temp. tokelvin ()
                                << "K" << endl;
return 0;
}
```

Abstraction

In c++, abstraction is achieved using:

1) Abstract classes (pure virtual functions)

2) Interfaces (only pure virtual functions)

3) Access specifiers (public, private, protected)

1) Abstraction using Abstract classes

An abstract class is a class that has at least one pure virtual function (a function declared with =0)
These classes cannot be instantiated and must be inherited by derived classes.

e.g:- Abstract class

```
# include <iostream>
using namespace std;

class Shape {
public:
virtual void draw() = 0;
virtual double area() = 0;
};
class circle : public shape {
private :
  double radius;
public:
circle (double r) : radius (r) { }
```

```
    void draw() override {
      cout <<"Drawing a circle"<<endl;
    }
    double area() override {
      return 3.14 * radius * radius;
    }
};
int main() {
Shape* s1 = new Circle(5);
Shape* s2 = new Rectangle(4,6);

s1->draw();
cout <<"Circle Area:" << s1->area() <<endl;
s2->draw();
cout <<"Rectangle Area:" << s2->area() << endl;
delete s1;
delete s2;
return 0;
}
```

2. Abstraction using Access specifiers

C++ access specifiers (private, protected, public) help in hiding unnecessary details.

e.g:-

```
# include <iostream>
using namespace std;

class BankAccount {

 private :
   double balance;

 Public :
 BankAccount (double initialBalance)
   {
    balance = InitialBalance;
   }
  void deposit (double amount)
   {
   balance += amount;
   cout <<" Deposited : " << amount << endl;
   }
void withdraw (double amount)
{ if (amount <= balance)
   {
    balance -= amount;
    cout <<" Withdrawn : " << amount << endl;
   }
   else {
     cout <<" Insufficient balance ! " << endl;
   }
}
```

```
    void displayBalance() {
        cout << "Current Balance : " << balance << endl;
    }
};

int main()
{
    BankAccount account(1000);
    account.deposit(500);
    account.withdraw(200);
    account.displayBalance();

    return 0;
}
```

Why is Abstraction Important?

- ✓ Reduces complexity
- ✓ Enhances maintainability
- ✓ Improves code Reusability.

3) Abstraction using Interfaces

C++ does not have a keyword for interfaces like Java, but we can simulate them using abstract classes with only pure virtual functions.

```
#include <iostream>
using namespace std;
class IAnimal {
  public:
    virtual void makeSound() = 0;
  };
class Dog : public IAnimal {
  public:
      void makeSound() override {
      cout << " Woof! Woof!" << endl;
      }
  };
class cat : public IAnimal {
  public:
      void makeSound() override {
          cout << " meow! meow! << endl;
        }
      };
int main() {
  IAnimal *pet1 = new Dog();
  IAnimal *pet2 = new cat();
  pet1->makeSound();
  pet2->makeSound();
  delete pet1;
  delete pet2;
  return 0;
}
```

Polymorphism

→ It is a core concept in C++ that allows objects of different types to be treated as objects of different types to be ~~th~~ treated as object of a common base type.

It enables a single interface to represent different underlying form (data types). There are two main types of polymorphism in C++.

1) Compile time Polymorphism (static Binding)

→ It is achived through function overloading and operator overloading.

→ function overloading : multiple functions with the same name but different parameters.

e.g :-

```
#include <iostream>
using namespace std;
void print (int i) {
    cout << "Integer : " << i << endl;
  }
void print (double d) {
    cout << " double : " << d << endl;
  }
```

```
void print(const char* str)
{
 cout<<"String :" << str << endl;
}
int main()
{
 print(5);
 print(3.14);
 print("Hello");
 return 0;
}
```

→ operator overloading : Define custom behavior for operators.

```
# include <iostream>
using namespace std;
class complex {
 public :
 int real, imag;
 complex(int r=0, int i=0) : real(r), imag(i) { }
```

(or)

```
/* complex(int r=0, int i=0) {
      this-> real = r;
      this-> imag = i;
 */ }
```

```
// overloading + operator
Complex operator +(const Complex &obj) {
    Complex res;
    res.real = real+obj.real;
    res.imag = imag+obj.imag;
    return res;
}

main()
{
Complex c1(3,4), c2(1,2);
Complex c3 = c1+c2;
cout <<"Result " <<c3.real <<" + " <<c3.imag <<" i" <<
                                                endl;

return 0;
}
```

2. Runtime Polymorphism (Dynamic Binding)

It is achived through inheritance and virtual functions.

Virtual functions : Allow derived classes to override methods in base class.

```
#include <iostream>
using namespace std;
class Animal {
 public:
    virtual void sound() {
    cout <<"Animal makes a sound" << endl;
    }
 };
class Dog : public Animal {
 public:
    void sound() override {
    cout << "Dog barks" << endl;
    }
 };
class cat : public Animal {
 public:
    void sound() override {
    cout << "cat meows" << endl;
    }
 };
int main() {
 Animal *a;
 Dog d;
 Cat c;
```

```
    a= &d;
    a-> sound();
    a= &c;
    a-> sound();
    return 0;
}
```

Virtual destructor

A virtual destructor in C++ is a destructor that is declared as virtual in the base class. It ensures that when an object is deleted through a base class pointer, the destructor of the derived class is also called, preventing memory leaks.

Why Use a Virtual Destructor?

When using polymorphism, if the base class destructor is not virtual, deleting a derived class object using a base class pointer will only call the base class destructor, leading to potential memory leaks.

e.g:- without a virtual Destructor (memory Leak)

```
#include <iostream>
using namespace std;
class Base {
  public:
   Base() { cout <<" Base Constructor\n"; }
   ~Base() { cout <<"Base Destructor\n"; }

class Derived : public Base {
 Public :
   Derived() {cout <<" Derived constructor\n"; }
 ~Derived() { cout <<"Derived Destructor\n"}
};
int main()
{
 Base *b = new Derived();
  delete b;
  return 0;
}
```

result:- issue: The Derived class destructor is not called, leading to a memory leak.

e.g:- with a virtual Destructor

```
#include <iostream>
using namespace std;
class Base {
 public:
 Base() { cout << " Base Constructor \n" ; }
 virtual ~Base() { cout << " Base Destructor \n" }
 };
class Derived : public Base {
 public:
 Derived() { cout << " Derived constructor \n" }
 ~Derived() { cout << " Derived destructor \n" }
 };
int main()
 {
 Base *b = new Derived();
 delete b;
 return 0;
}
```

Key Points:-

1) A virtual destructor ensure proper destruction of derived class objects when deleted via a base class pointer.

2) If a base class has any virtual function, its destructor sould als be virtual.

3) Virtual destructors prevent memory leaks in polymorphic classes.

4) They introduce a slight runtime overhead due to dynamic dispatch, but the benefits out weigh the cost.

Inheritance

It allows one class (child/derived) to acquire the properties & behaviour of another class (parent/base). This promotes code reusability & hierarchical classification

Types of Inheritance in C++

1) single Inheritance
2) multiple Inheritance
3) multilevel Inheritance
4) Hierarchical Inheritance
5) Hybrid (virtual) Inheritance

1) single Inheritance
A derived class inherits from a single base class.

```
#include <iostream>
using namespace std;
class Parent {
public :
    void show() { cout <<" This is the parent class" << endl;
    }
};
```

```
class child : public Parent {
public :
    void display()
    {
     cout <<"this is the child class" << endl;
    }
};
int main()
{
 Child obj;
 obj. show();
 obj. display();
  return 0;
}
```

2) Multiple Inheritance

A derived class inherits from more than one base class.

```
#include <iostream>
using namespace std;
class A {
public:
  void showA() { cout <<"class A" << endl; }
  };

class B {
 public:
   void showB() { cout<<"class B" << endl;
   };
```

```
class c {
  public :
void show{cout<<" class c"<<endl;}
  };

class multiple : public A, public B, public c {
  public :
void showMultiple ()
  {
    cout<<" multiple : "<<endl;
  }
};

int main()
  {
  multiple m ;
    m.showA();
    m.showB();
    m.show c();
  }
```

3) multilevel Inheritance

A derived class inherits from another derived class, forming a chain.

```
# include <iostream>
using namespace std;
class Grandparent {
  public :
void showG() {cout <<" Grandparent class"<< endl; }
  };
```

3) Multi level Inheritance

A derived class inherits from another derived class, forming a chain.

```
#include <iostream>
using namespace std;
class Grandparent {
 Public :
  void showG() { cout <<"Grandparent" << endl; }
   };
class parent : public Grandparent {
 public :
 void showP() { cout <<" parent " <<endl; }
   };
class child : public Parent {
 public :
  void showC() { cout <<" child" <<endl; }
  };
int main()
 {
 child obj;
 obj. showG();
 obj. showP();
 obj. showC();
 return 0;
}
```

4) Hierarchical Inheritance

multiple derived classes inherit from a single base class.

```
# include <iostream>
using namespace std;
class parent {
  public :
  void show() { cout <<" parent class"<< endl; }
  };
class child1 : public Parent {
  public :
      void show1() {cout <<" child1 class" <<endl; }
  };
class child2 : public parent {
  public:
  void show2() { cout <<" child2 class "<<endl; }
  };
int main()
{
child1 obj1;
child2 obj2;
obj1.show ();
obj1.show1();
obj2.show();
obj2.show2();
return 0;
}
```

5) hybrid (virtual) Inheritance

hybrid Inheritance is a combination of multiple types of Inheritance, such as multiple inheritance, hierarchical inheritance, and multilevel inheritance. It is used to design complex class hierarchies while maintaining code reusability and organization.

However, hybrid inheritance can introduce the "diamond problem", which occurs when multiple paths lead to a common base class. This problem can be resolved using virtual inheritance.

e.g:- combination of multiple & Hierarchical Inheritance.

```
#include <iostream>
using namespace std;
class A {
  public:
    void showA() {
      cout <<"class A" <<endl;
    }
};

class B: public A {

Public:
    void showB() {
    cout <<"class B" <<endl;
  }
};
```

```
class C : public A {
public :
  void showC() {
    cout <<"Class C" << endl;
  }
};
class D : public B, public C {
public :
  void showD() {
    cout <<"cout D" << endl;
  }
};
int main()
{
  D obj;
  obj. showB();
  obj. showC();
  obj. showD();
  // obj. showA(); // error : Ambiguity due to multiple inheritance
  return 0;
}
```

problem : Ambiguity in Base class members

In the above example, class D inherits from both B & C, which both inherit from A. This leads to ambiguity when calling showA() as D has two copies of A.

e.g:- Solving the Diamond problem using Virtual Inheritance.

To avoid ambiguity, we use virtual inheritance while inheriting from A:

```
#include <iostream>
using namespace std;
class A {
 public:
  void showA() {
    cout <<"class A" <<endl;
  }
};
class B : virtual public A {
public:
 void showB() {
   cout<<"class B"<<endl;
 }
};
class C : virtual public A {
public :
  void showC() {
  cout <<"class C" <<endl;
  }
};
```

```
class D : public B, public C {
 public :
   void showD() {
    cout << " Class D" << endl;
   }
 };

int main()
 {
  D obj;
  obj.showA(); // No Ambiguity Now
  obj.showB();
  obj.showC();
  obj.showD();

  return 0;
 }
```

Key points of Hybrid Inheritance

→ Combines different types of Inheritance (multiple, multilevel, Hierarchical)

→ Diamond problems occurs when a derived class inherits multiple times from a common base class.

→ virtual inheritance prevents duplicate copies of the base class.

→ It is used in complex system where classes have multiple functionditics.

When to use Hybrid Inheritance?

- When designing systems with a mix of functionalities from different sources.
- → when multiple base classes contribute to the same derived class
- → when managing complex relationships in object hierarchies (e.g:- real-world models like employees, vehicles etc.)

~~when would~~

How many types of constructor in c++?

A constructor is a special member function in c++ that gets called automatically when an object of a class is created. It is mainly used to Initialize objects.

Key features of constructors

→ Same name as the class
→ No return type (Not even void)
→ called automatically when an object is created.
→ can be overloaded

Types of constructors in c++

1) Default constructor (No parameters)

A constructor that takes no arguments and initializes the object with default values.

```
#include <iostream>
using namespace std;
class Car {
public:
  car() {
  cout<<"Car Created"<<endl;
  }
};
int main()
{
Car c1;
return 0;
}
```

O/P

car created

2) Parameterized Constructor

A constructor that accepts parameters to initializes an object.

```
#include <iostream>
using namespace std;
class Car {
 public:
 string brand;
 int speed;
 Car (string b, int s)
 {
  brand = b;
  speed = s;
 }
 void display () {
 cout <<" Brand " << brand << endl;
  }
};
int main()
{
Car c1 ("Tesla", 200);
 c1. display();
 return 0;
}
```

3) Copy constructor

A constructor that creates a copy of another object.

```
# include <iostream>
using namespace std;
class Car {
 Public:
 string brand;
 int speed;
 car (string b, int s) {
    brand = b;
    speed = s;
 }
 car (const car &c) {
  brand = c.brand;
  speed = c.speed;
 }
 void display() {
  cout <<" Brand : " << brand << endl;
 }
};

int main()
{
 car c1("BMW", 180);
 car c2 = c1;
 c2.display();
 return0;
}
```

4) Destructor (~)

A destructor is used to release resources before an object is destroyed. It has the same name as the class but with a tilde (~) before it.

```
#include <iostream>
using namespace std;
class car {
public:
car() {
  cout <<" car Created !" <<endl;
  }

~car() {
  cout <<"car Destroyed !" << endl;
  }
};
int main()
{
car c1;
return 0;
}
```

5) constructor overloading.

multiple constructors in the same class with different parameters.

```
#include <iostream>
using namespace std;
class car {
 public:
 string brand;
 int speed;

 car() {
 brand = "unknown";
 speed = 0;
 }
 car (string b, int s)
 {
  brand = b;
  speed = s;
 }
 void display() {
  cout << " Brand : " << brand << endl;

  }
 };
int main()
{
car c1;
car c2 (" Audi ", 220);

c1.display();
c2. display();
}
```

6) constructor with Dynamic memory allocation

when using pointers & dynamic memory (new), always define a destructor to avoid memory leaks

```
#include <iostream>
using namespace std;

class Car {
public:
    int *speed;

  Car (int s) {
    speed = new int(s);
  }
  ~Car() {
    delete speed;
    cout <<" memory freed !" << endl;
  }
void display() {
    cout <<"speed : " << *speed << " km/h" << endl;
  }
};
int main()
{
Car c1(150);
c1.display();
return 0;
}
```

Interview Questions on Constructors

1) what is a constructor in c++?
2) why is the copy constructor needed?
3) what happens if you don't define a constructor?
4) what is the difference between a constructor & a destructor?
5) what is constructor overloading?
6) why do we use a virtual destructor in c++?

summary :-

Type	purpose
Default constructor	→ Initializes object with default values
parameterized constructor	→ Initializes object with user-defined values
copy constructor	→ copies values from one object to another
constructor overloading	→ multiple constructors with different parameters.
Destructor (~ClassName)	→ clean up memory when object is destroyed.

Interview Questions on Class & object

Q1) what is a constructor in c++? why do we need it?

→ A constructor is a special member function in c++ that is automatically called when an object of a class is created.

Key points:

→ The constructor name must be the same as the class Name.

→ It has no return type (Not even void)

→ It is used to Initialize objects of a class

→ It can takes arguments (this is called a parametorized constructor).

e.g:-

```
#include <iostream>
using namespace std;
class car {
 public:
  car() {
   cout <<" car object created" << endl;
   }
 };
int main()
{
 car c1;
 return 0;
}
```

Q2) Can a constructor have a return type? why or why not?

→ A constructor's purpose is to initialize an object, not to return a value.

→ If a constructor had a return type, it would behave like regular function & not a constructor.

→ The compiler expects constructors to have no return type. if you try to specify one, it will result in an error.

Example: (Invalid code)

```
class car {
  public:
   int car () {
      return 10;
    }
  };
```

Note:- compilation Error: constructor cannot have a return type.

Q3) when is a constructor called?

→ A constructor is automatically called when an object of a class is created, whether:

1) without parameters (default constructor)

2) with parameters (parameterized constructor)

3) Using another object (copy constructor)

Example 1: Default constructor

```
#include <iostream>
using namespace std;
class car {
 public :
  car() {
  cout <<" car constructor called!"<< endl;
  }
};
int main()
 {
 Car c1;
  return 0;
 }
```

Example 2: Parameterized constructor

```
class car {
 public:
   car (string name) {
   cout <<" car name : "<<name<<endl;
   }
};
int main()
 {
 car c1 ("Tesla");
  return 0;
 }
```

Q3) what is the difference between a constructor & a destructor?

Feature	Constructor	Destructor
Purpose	Initialize an object	Cleans up resources before object is destroyed
Name	Same as the class name	Same as class name, but prefixed with ~
Return type	No return type	No return type
Parameters	Can have parameters (parametorized constructor)	No parameters allowed
overloading	can be overloaded	cannot be overloaded (only one per class)
Automatic class	called when an object is created	called automatically when an object goes out of scope or it is deleted.

e.g:-

```
#include <iostream>
using namespace std;
class car {
public:
  car() {   // constructor
   cout <<" constructor : car object created !" <<endl;
  }
~car() {
 cout <<" Destructor : car object destroyed ! " <<endl;
  }
};
```

```
int main()
{
 car c1;
 return 0;
}
```

O/P

constructor : car object created!

Destructor : car object destroyed!

Q4) can we overload a constructor in c++ ? How?

→ yes, constructor overloading works just like function overloading, but the constructor name remains the same while the parameters differ.

e.g:- constructor overloading

```
# include<iostream>
using namespace std;
class car {
public:
 car() {
 cout <<" Default constructor : No parameters" << endl;
 }
car (string name) {
cout <<" parameterized constructor : car name is "
        <<name <<endl;
 }
```

```
    //Another parameterized constructor
    car (string name)
    {
    cout<<" parameterized constructor : car name is "<< name<<endl;
    }
    car (string name, int year)
    {
    cout <<" overloaded constructor : "<< name << "was made in"
          << year <<endl;
    }
};
int main()
{
  car c1;
  car c2("Tesla");
  car c3("BMW", 2023);
  return 0;
}
```

o/p

Default constructor : No parameters

parameterized constructor : car name is Tesla

overloaded constructor : BMW was made in 2023

Note :-

1) same name but different parameters → constructor overloading
2) Helps in flexible object initialization
3) Automatically calls the right constructor based on arguments passed.

Q5) What is copy constructor?

→ A copy constructor is a special constructor in C++ that creates a new object as a copy of an existing object.

Syntax:-

```
ClassName (const ClassName &oldobject);
```

→ It takes a reference (&) to an existing object of the same class.

→ It copies all member variables from the existing object to the new object.

Q6) When is a copy constructor called?

A copy constructor is called automatically when:

i) An object is initialized from another object of the same class.

ii) An object is passed by value to a function

iii) An object is returned by value from a function.

e.g:-

```
#include <iostream>
using namespace std;
class car {
 public:
   string name;
   car (string n) {
   name=n;
   }
```

```
car (const car &oldcar)
  {
  name = oldcar.name;
  cout<<" copy constructor called !" <<endl;
  }
};

int main()
{
car car1("Tesla"); // Normal constructor
car car2 = car1; // copy constructor is called
cout <<"Car 1 Name : " <<car1.name<<endl;
cout <<"Car 2 Name : " <<car2.name <<endl;

return 0;
}
```

Note:-

i) shallow copy : Default copy constructor copies only values (Not memory allocation)

ii) Deep copy : If an object has dynamic memory, you need a custom copy constructor to allocate new memory. (otherwise, changes in one object affect the other).

Q7) what is the difference between a shallow copy & deep copy ?

→

Feature	Shallow copy	Deep copy
Defination	Copies the memory address of an object	Create a new object and copies the actual data.
Copy Type	Default copy constructor	Requires custom copy constructor & dynamic memory allocation.
memory usage	uses the same memory as the original object (shared data)	allocates seperate memory for the new object
Data safety	Modify one object affects the other (because they share memory)	Modifying one object does not affect the other
use case	suitable for simple objects without dynamic memory allocation	Necessary when an object contain pointers or dynamic memory.

eg:- shallow copy

```
# include <iostream>
using namespace std;
class shallow {
public :
 int * data;
 shallow (int value) { // constructor
  data = new int (value);
 }
 ~shallow () { // Destructor
  delete data;
 }
};

int main()
{
shallow obj1(10);
Shallow obj2 = obj1; // Default copy constructor (shallow copy)
cout << *obj1.data << endl;
cout << *obj2.data << endl;
delete obj1.data; // Delete share memory
cout << *obj2.data; // undefined behaviour

 return 0;
}
```

Note:- when obj1 is deleted, obj2 also loses access to memory, causing dangling pointer issues.

e.g:- Deep copy

```
#include <iostream>
using namespace std;
class Deep {
public:
  int *data;
Deep (int value) { // constructor
data = new int (value);
}
Deep (const Deep& obj) {
data = new int (*obj.data);
}
~Deep() {
  delete data;
}
};
int main()
{
Deep obj1 (10);
Deep obj2 = obj1;  // calls deep copy constructor

cout << *obj1.data << endl;
cout << *obj2.data << endl;
delete obj1.data; // only obj1 memory is freed, obj2 remains safe.
cout << *obj2.data;
return 0;
}
```

solution:- Each object gets its own copy of data, preventing shared memory issues.

Note:-

1) shallow copy copies memory addresses, ~~being~~ leading to dangling pointers.

2) Deep copy creates separate memory allocations, making the objects independent

3) Always use a custom copy constructor for deep copying when dealing ~~when~~ with dynamic memory allocation (pointers).

Q8) what is a virtual destructor in c++? why do we need it?

→ virtual Destructor in c++

A virtual destructor ensures that when an object is deleted through a base class pointer, the derived class destructor is also properly, preventing memory leaks.

→ ~~ω~~ If we delete a derived class object using a base class pointer, and the base class does not have a virtual destructor, the derived class destructor will not be ~~cla~~ called, leading to a memory leak.

e.g.:- without a virtual Destructor (memory Leak)

```
#include <iostream>
using namespace std;
class Base {
  public :
    Base() { cout<<" Base constructor\n" }
  ~Base() { cout<<"Base Destructor\n" } // Not virtual
};
```

```
class Derived : public Base {
Public :
    Derived() { cout <<" Derived constructor\n" ; }
  ~Derived() { cout <<" Derived Destructor\n" ; } // won't be called
};
int main() {
  Base* ptr = new Derived();
  delete ptr;
  return 0;
}
```

O/P :-

Base Constructor

Derived constructor

Base Destructor X (Derived Destructor is NOT called!)

→ problem : The derived class destructor is skipped, causing a memory leak.

eg :- Example with a virtual Destructor (correct cleanup)

```
#include <iostream>
using namespace std;
class Base {
public:
Base() { cout <<" Base constructor\n" ; }
virtual ~Base() { cout <<" Base constructor\n" }; // virtual Destructor
};
```

```
class Derived : public Base {
public:
  Derived() { cout<<"Derived constructor\n"; }
  ~Derived() { cout<<"Derived Destructor\n"; } // Now it will be
                                                  called
};
int main() {
  Base * ptr = new Derived(); // Base pointer to Derived object
  delete ptr; //calls both base & Derived destructors
  return 0;
}
```

o/p

Base constructor
Derived constructor
Derived Destructor (proper cleanup)
Base Destructor

Sol^n:- The virtual destructor ensures that both destructors are called, preventing memory leaks.

Note:-

→ If a class has virtual functions, it should have a virtual destructor to ensure proper cleanup

→ without a virtual destructor, deleting a derived object via a base class pointer causes memory leaks.

→ virtual destructors allow proper cleanup in inheritance hierarchies.

Q9) What is a pure virtual function in C++?

A pure virtual function is a function that must be overriden by a derived class. It is declared in a base class but has no implementation.

A class that contains at least one pure virtual function is called an abstract class and cannot be instantiated.

Syntax:-

```
class Base {
public:
 virtual void show() = 0; // pure virtual function
};
```

e.g:- pure virtual function

```
#include <iostream>
using namespace std;
class Shape {
public :
 virtual void draw() = 0; // pure virtual function (No Implement
};
class Circle : public shape {
public:
 void draw() override { // must ~~Declare~~ override
   cout<< "Drawing a circle\n" ~~<<endl~~;
   }
};
```

```
int main()
{
 //shape obj ; //Error: ~~cannot~~ cannot create an object of an abstract class.

 Circle c;
 c.draw(); //calls overriden functions

 return 0;
}
```

O/P

drawing a circle.

Note:- pure virtual function (draw) is overriden in the derived class. (circle).

i) pure virtual functions force derived classes to implement them.

ii) Abstract classes cannot be instantiated.

iii) used in ~~Polymorphism~~ polymorphism to create a common interface.

Q10) what is an interface in c++?

→ An interface in c++ is a class that contain only pure virtual functions. It is used to enforce a contract, ensuring that derived classes must implement specific methods.

How to create an Interface in c++?

An interface is ~~used~~ crated using an abstract class with only pure virtual functions.

e.g:-

```
class Interface {
public:
virtual void show() = 0; // pure virtual function
virtual void display() = 0; // Another pure virtual function
```

→ This class cannot have object instances.

→ Any class inheriting this interface must implement all its functions.

e.g:- Interface in C++

```
#include <iostream>
using namespace std;
class Ishape {
public:
virtual void draw() = 0;    // pure v. f.
virtual double area() = 0; // Another p.v.f
};
class Circle : public Ishape {
double radius;
public:
circle (double r) : radius (r) { }
void draw() override {
cout<<"Drawing a circle\n";
}
double area() override {
return 3.14 * radius * radius;
}
};
```

```
int main()
{
 Circle c(5.0);
 c.draw();
 cout<<"Area : "<< c.area() << endl;
  return 0;
 }
```

O/P

Drawing a circle

Area : 78.5

The circle class implements all pure virtual functions, making it a concrete class.

Note:-

i) Interfaces in c++ are created using abstract classes with only pure virtual functions.

ii) Derived classes must implement all pure virtual functions of the interfaces.

iii) Interfaces enable polymorphism and multiple inheritance.

Q10) what is the difference between an abstract class and an interface?

Features	Abstract class	Interface
i) Defination	A class that can have both normal & pure virtual functions.	A class that contains only pure virtual functions.
ii) Purpose	provides partial implementation that can be shared among derived classes.	Enforced a strict contract for derived classes.

	abstract class	Interface
iii) function types	A class that can have both normal & pure virtual functions.	A class that contains only pure virtual functions.
iv) purpose	provides partial implementation that can be shared among derived classes.	Enforces a strict contract for derived classes.
v) function types	can have both normal functions and pure virtual functions	only pure virtual function (No implementation)
vi) Data Members	can have member variables	cannot have any data members.
vii) Constructor & Destructor	can have constructors & destructors.	can have a virtual destructor but no constructors.
viii) multiple Inheritance	supports single & multiple Inheritance	used for multiple inheritance in C++.

e.g:- abstract class (Has Implementation)

```
#include <iostream>
using namespace std;
class AbstractShape {
Public:
void commonfunction() { //Normal function allowed
  cout <<" Common function in Abstract class\n";
}
virtual void draw() = 0; //Pure virtual function
};
```

```
class Circle : public AbstractShape {
  public:
  void draw() override {
   cout <<" Drawing a circle\n";
   }
};
int main()
{
 Circle c;
 c.commonfunction();  // Allowed
 c.draw();
  return 0;
 }
```

O/P

common function in Abstract class

Drawing a circle

Note :- The abstract class allows both normal & pure virtual functions.

e.g.:- Interface (only pure virtual functions)

```
#include <iostream>
using namespace std;

class Ishape {
 public:
    virtual void draw() = 0; // only pure virtual functions
    virtual double area() = 0;
};

class Rectangle : public Ishape {
    double length, width;

 public:
  Rectangle (double l, double w) : length (l), width (w) { }
  void draw() override {
   cout<<" Drawing a Rectangle\n";
  }
  double area() override {
     return length * width;
   }
 };
int main()
{
Rectangle r(5.0, 3.0);
r.draw();
cout<<" Area: "<< r.area() <<endl;
return 0;
}
```

Note:- Ther interfaces ensures all functions must be implemented in the derived class.

i) Abstract class = can have normal & pure virtual functions.

ii) Interface = Has only pure virtual functions (acts as a contract)

iii) use an abstract class when you need common functionality among derived classes.

iv) use an interface when you need strict enforcement of function implementation.

Q11) what is the "Diamond problem" in c++ multiple inheritance?

The Diamond problem occurs in multiple inheritance when a class inherits from two classes that both inherit from a common base class. This can lead to ambiguity in function / method calls.

e.g:-

```
#include <iostream>
using namespace std;
class A {
public:
  void show() {
    cout <<"class A\n";
  }
};
class B : public A { }
class C : public A { }
class D : public B, public C { };
```

```
int main()
{
  D obj;
  obj.show(); //Ambiguity
  return 0;
}
```

Note:- since D inherits from both B & C, and both classes inherits from A, the compiler does not know whether to call A::show() from B or C.

Solution: use virtual Inheritance

we solve this issue by using virtual inheritance, ensuring that only one copy of A exists in D.

```
#include <iostream>
using namespace std;
class A {
Public:
 Void show() {
  cout << "class A\n";
  }
};
Class B : virtual public A { };
class C: virtual public A{ };
class D : public B, public C { }; // No abigeuity Now
int main()
{
 D obj;
obj.show();
return 0;
}
```

Note:- using virtual ensures that only one copy of A exists, avoiding ambiguity.

i) The Diamond problem occurs in multiple inheritance when two parent classes share the same base class.

ii) It causes ambiguity when calling methods of base class.

iii) Virtual Inheritance prevents multiple copies of the base class & resolves ambiguity.

Q12) what is a virtual function and why do we need it?

→ A virtual function is a member function in a base class that is overriden in a derived class and is resolved at runtime using dynamic (late) binding.

By default, C++ uses early binding (function calls are resolved at compile time). A virtual function allows us to overriden this behavior, ensuring that the function from the derived class is called instead of the base class version.

why do we need virtual function?

→ To achive runtime polymorphism

→ Allows overriding a function in derived classes.

→ Helps implement dynamic method dispatch (correct function is selected at runtime)

e.g:- without virtual function

```
#include <iostream>
using ~~na~~ namespace std;
public:
  void show() {
  cout <<" Base class function\n";
  }
};
```

```
class Derived : public Base {
public:
    void show()
    {
    cout << " Derived class function << endl;
    }
};
int main () {
 Base * ptr;
 Derived obj;
 ptr = & obj;
 ptr -> show(); // calls : Base:: show() (Early Binding)
  return 0)
}
```

problem: The function from the base class is called instead of the derived class.

e.g:- using virtual function (Late Binding)

```
# include <iostream>
using namespace std;
class Base {
 public :
 virtual void show() {
  cout << " Base class function" << endl)
  }
};
```

```
class Derived : public Base {
 public:
    void show() & override {
      cout<<"Derived class function <<endl;
    }
  };

int main() {
  Base *ptr;
  Derived obj;
   ptr = & obj;
   ptr -> show(); // calls Derived::show() (Late Binding)
    return 0;
  }
```

Note:- since show() is a virtual function, the function from Derived is called at runtime.

i) without virtual, c++ uses early binding (calls base class function)

ii) with virtual, c++ uses late binding (calls derived class function)

iii) virtual function allow polymorphism, enabling dynamic method dispatch.

Q13) what is a pure virtual function ? How is it different from a virtual function ?

→ A pure virtual function is a virtual function that must be overriden by a derived class. It is declared using = 0. in the base class. making the class abstract

Difference Between virtual & pure virtual Function

Feature	virtual function	pure virtual function
Defination	Defined in base class	Declare in base class with = 0
Implementation	can have a definition	cannot have a defination (forces derived classes to override it)
purpose	Allows overriding but not mandatory	forces overriding in derived class.
Base class	can be used in normal class	makes the class abstract

e.g:- pure virtual function

```
#include <iostream>
using namespace std;
class shape {
public:
virtual void draw () = 0;
};
```

```
class Circle : public Shape {
 public:
    void draw() override {
      cout <<"Drawing circle <<endl;
    }
 };

class Rectangle : public Shape {
 public :
  void draw() override {
   cout << " Drawing Rectangle <<endl;
   }
 };

int main()
 {
 Shape *s1 = new circle ();
 Shape *s2 = new Rectangle();

 s1 -> draw();
 s2 -> draw();

 delete s1;
 delete s2;
 return 0;
}
```

Note:-

i) shape is an abstract class because it contains a pure virtual function.

ii) Any class that inherits shape must implement draw(), or it will also be abstract.

iii) The draw() function is overriden in Circle & Rectangle.

iv) A class with at least one pure virtual function is an abstract class

v) we can not create objects of an abstract class.

vi) Derived classes must override pure virtual functions.

vii) Used when we want to enforce a specific interface across multiple derived classes.

Q14) What is an Abstract Class? How is it different from a Normal class?

→ An abstract class is a class that cannot be instantiated and is used as a base class for other classes.
It contains at least one pure virtual function (=0).

Feature	Normal class	Abstract class
Instantiation	can create objects	cannot create objects.
Purpose	used for defining objects	used for enforcing a structured in derived classes.
Contains pure virtual function?	NO	yes (at least one)
Derived class	may override base functions	must override all pure virtual functions.

e.g:- abstract class

```
#include <iostream>
using namespace std;
class Animal {
public :
  virtual void sound() = 0;
};
class Dog: public Animal {
public :
  void sound() override {
    cout<<"Dog barks"<<endl;
  }
};

class Cat : public Animal {
public :
  void sound() override {
    cout<<"cat meows" <<endl;
  }
};
int main() {
  // Animal a; → error : can not instantiate abstract class
  Animal *a1 = new Dog();
  Animal * a2 = new cat();
  a1 -> sound();
  a2 -> sound();
  delete a1;
  delete a2;
  return 0;
}
```

Explaination:-

→ Animal is an abstract class because it contains a pure virtual function (sound()).

→ Dog & cat must override sound(), or they will also be abstract.

→ we cannot create objects of Animal directly.

Key points:

i) Abstract classes cannot have objects.

ii) Used as a blueprint for derived classes.

iii) Derived classes must override all pure virtual functions.

iv) Helps in achieving polymorphism.

Q15) what is Difference between Abstract class & Interface?

C++ does not have interfaces like JAVA, but we can achieve the same functionality using pure virtual functions in an abstract class.

Feature	Abstract class	Interface
purpose	provides partial implementation	provides a complete blueprint (no implementation)
Functions	can have normal & pure virtual function	only pure virtual functions (=0).
Data members	can have member variable	Should not have member variable (only function prototypes)
Constructor	can have a constructor	cannot have a constructor

multiple Inheritance	Supports single & multiple inheritance	used to simulate multiple Inheritance
use case	Used when some implementation is shared.	Used when only method declaradions are needed

example :- Abstract class (partial implementation)

```
#include <iostream>
using namespace std;
class Animal {
 public:
  virtual void sound()=0;
  void sleep() {
  cout<<"Animal is sleeping"<<endl;
  }
};

class Dog : public Animal {
 public :
  void sound() override {
    cout<<" Dog barks"<<endl;
  }
};
int main() {
 Dog d;
 d.sound();
 d.sleep();
 return 0;
}
```

Explanation:-

-> Animal has both pure virtual (sound()) & normal (sleep()) functions.

-> Dog must override sound(), but it inherits sleep() without overriding it.

Q16) what is the difference between Deep copy & shallow copy ?

-> shallow copy

-> copies only the memory address (not the actual data)

-> If one object is deleted, the other object's pointer becomes dangling, leading to undefined behavior.

-> Default copy constructor does a shallow copy.

```
#include <iostream>
# include <cstring>
using namespace std;
class shallow {
public:
 char* name;

  //constructor
  shallow (const char* str) {
  name = new char [strlen(str) + 1]; // Allocating heap memory
   strcpy (name, str);
  }
```

```
shallow (const shallow& obj) {
    name = obj.name; // only copying address not actual data
  }

  // Destructor

~ shallow () {
    delete[] name;
  }
  void display () {
    cout<<"Name : "<<name<<endl;
  }
};

int main ()
  {
  shallow obj1 ("Rahul");
  shallow obj2 = obj1;
   obj1. display();
   obj2. display();
   return 0;
  }
```

issue is shallow copy

→ when obj 1 is destroyed, it deletes the memory.

→ since obj2 has the same address, accessing name in obj2 l

to undefined behaviour (dangling pointer)

→ Deep copy.

```
#include <iostream>
#include <cstring>
using namespace std;
class Deep {
 public:
 char *name;
Deep (const char* str) {
name = new char [strlen (str) +1];
 strcpy (name, str);
}
//copy constructor (Deep copy)
Deep (const Deep& obj)
{
name = new char [strlen (obj.name) +1]; // Allocate new memory
strcpy (name, obj.name);
}
//Destructor
~Deep() {
delete [] name;
}
void display() {
cout <<"Name : " << name << endl;
}
};
```

```
int main()
{
Deep obj1("Rahul");
Deep obj2 = obj1;
obj1.display();
obj2.display();
return 0;
}
```

why Deep copy is safe

-> Each object has its own copy of data.

-> No memory sharing -> prevents dangling pointer issues.

-> when obj1 is destoryed, obj2 still holds a valid copy.

Key Differences:

Features	shallow copy	Deep copy
what is copies?	memory Address	Actual Data
memory sharing?	yes	NO
Risk?	Dangling pointer issue	safe
Performance	faster	Slower
Default Behavior	Yes	NO

Types of pointers in C++

→ pointers are special variable that store memory addresses.

In C++, there are different types of pointers based on their use cases. Let's explore them one by one with examples.

1) NULL Pointer

A pointer that is initialized with nullptr (or NULL in older C++ version) used to avoid dangling pointers.

ex:-

```
#include <iostream>
using namespace std;
int main() {
  int * ptr = nullptr;
  if (ptr == nullptr)
    {
     cout<< "pointer is NULL" <<endl;
    }
  return 0;
}
```

Problem with Normal pointers

Some Issues with normal pointer in C++ are as follows:

→ 1) Memory Leaks : This occurs when memory is repeatedly allocated by a program but never freed. This leads to excessive memory consumption & eventually leads to a system crash.

) Dangling pointers: A dangling pointer is a pointer that occurs at the time when the object is de-allocated from memory without modify the value of the pointer.

i) wild Pointers: wild pointers are pointers that are declared and allocated memory but the pointer is never initialized to point to any valid object or address.

v) Data Inconsistency: Data inconsistency occurs when some data is stored in memory but is not updated in a consistent manner.

v) Buffer overflow: when a pointer is used to write data to a memory address that is outside of the allocated memory block. This leads to the corruption of data which can be exploited by malicious attackers.

e.g :-

```
# include <iostream>
using namespace std;
class Rectangle {
private :
    int length;
    int breadth;
};
void func()
{
Rectangle * p = new Rectangle();
}
```

```
int main()
{
  while(1)
  {
    func();
  }
}
```

o/p :-

memory limit exceeded

~~Explanation :- In function fun.~~

Smart pointers

As we've known unconsciously not deallocating a pointer causes a memory leak that may lead to a crash of the program.

Languages Java, C# has Garbage Collection mechanisms to smartly deallocate unused memory to be used again.

The programmer doesn't have to worry about any memory leak

C++ comes up with its own mechanism that's smart pointer.

when the object is destroyed it frees the memory as well.

so we don't need to delete it as smart pointer will handle it.

e.g:- demonstrate the working of smart pointer

```
#include <iostream>
using namespace std;
class SmartPtr {
  int * ptr; // Actual pointer
 public:
  explicit smartptr (int *p=NULL)
     {
      ptr = p;
     }
  ~SmartPtr () {
   delete (ptr);
  }
 int& operator*()
 {
  return *ptr;
 }
};
int main()
 {
 SmartPtr ptr(new int());
 *ptr = 20;
 cout << *ptr;

 return 0;
 }
```

Note:- we don't need to call delete ptr; when the object ptr goes out of scope, the destructor for it is automatically called & destructor does delete ptr.

Smart pointers in C++ help manage dynamically allocate memory automatically, preventing memory leaks and dangling pointers. They are available in the <memory> header.

Types of smart pointers in C++

Smart Pointer	Description
1) unique_ptr	owns & manages a single object. Transfer ownership
2) shared_ptr	Allows multiple shared ownership of the same object
3) weak_ptr	works with shared_ptr, avoids circular reference

1) unique_ptr - Exclusive ownership

unique_ptr ensures only one pointer owns a resource at a time.

```
# include <iostream>
# include <memory>
class MyClass {
 Public :
 myClass (int val) { std::cout <<" constructor called with value:"
                    << val << std::endl;

 ~myclass() {
  std::cout <<" Destructor called" << std::endl;
  }
void show() {
 std::cout <<" myClass function " << std::endl;
}; };
```

```
int main()
{
std::unique_ptr<myClass> ptr1 = std::make_unique<myClass>(10);
                                    //smart allocation

  ptr1->show();

  //ownership Transfer

  std::unique_ptr<myclass> ptr2 = std::move(ptr1);
   if (!ptr1)
   {
   std::cout<<" ptr1 is now nullptr"<< std::endl;
   }
   return 0;
  }
```

Key features:

- No two unique_ptr objects can own the same resource.
- Supports move semantics (std::move())
- Automatically destroy the object when out of scope.

2) Shared_ptr - Shared ownership

Shared-ptr allows multiple smart pointers to share the same object.

```
#include <iostream>
#include <memory>

class myClass {

public:
myClass (int val) { std::cout<<"Constructor : " <<val<<std::endl;
                  }

~myclass () {
  std::cout <<"Destructor called"<<std::endl;
  }

int main()
{
std::shared-ptr<myClass> sp1 = std::make_shared<myclass> (100);

std::shared-ptr<myClass> sp2 = sp1;

std::cout<<" Reference count"<<sp1.use_count() << std::endl;

  return 0;
}
```

Note:-

-> uses reference counting (use-count())

-> object is destroyed when all shared-ptr instances go out of scope.

3) weak_ptr - Non-owning Reference

weak_ptr is used to avoid circular references in shared_ptr.

```
#include <iostream>
#include <memory>

class myclass {
 public :
  myClass () {
   std:: cout <<" Constructor called" << std:: endl;
    }
  ~myclass () {
   std:: cout <<" Destructor called " << std:: endl;
   }
  };
int main() {
std:: shared_ptr <myClass> sp = std:: make_shared <myClass> ();

std:: weak_ptr<myClass> wp = sp;

std:: cout <<" use count " << sp. use_count() << std:: endl;

if (auto temp = wp.lock())
    {
    std:: cout <<" object is still alive" << std:: endl;

    } else
      {
      std:: cout <<" object is destroyed" << std:: endl;
      }

   return 0;

 }
```

Note:-

- weak_ptr does not increase reference count.
- Used to check if an object still exist (lock() function).

Q1) when to use which smart pointer?

smart pointer	use case
unique_ptr	when ownership must be exclusive (e.g:- resource management)
shared_ptr	when multiple pointer need shared ownership (e.g:- graphs, trees)
weak_ptr	when avoiding circular reference with shared_ptr.

Q2) what is the difference between unique_ptr & shared_ptr?

Feature	unique_ptr	shared_ptr
Ownership	single ownership	shared-ownership
copyable?	No	yes
Reference count	No reference counting	uses reference counting (use_count())
memory Release	object is deleted when unique_ptr goes out of scope	object is delete only when last share_ptr goes out of scope.
performance	Faster	slightly slower

e.g:-

```
std::unique_ptr<int> uPtr1 = std::make_unique<int>(10);
std::shared_ptr<int> sPtr1 = std::make_shared<int>(20);
//std::unique_ptr<int> uPtr2 = uPtr1; // Error: copy not allowed
std::shared_ptr<int> sPtr2 = sPtr1; // sPtr1 & sPtr2 share ownership
```

Q3) How does weak_ptr help prevent circular reference?

→ Circular Reference problem.

If two shared_ptr objects reference each other, reference count never reaches zero, causing a memory leak.

Sol^n:-

→ weak_ptr provides a non-owning reference to an object managed by shared_ptr.

→ It does not increase the refference count, preventing circular dependency.

E.g:- Circular Reference Issue:

```
#include <iostream>
#include <memory>

class B; // forward declaration

class A {
public:
std::shared_ptr<B> bptr;
A() {std::cout << "A created" << endl;
~A() {std::cout << "A destroyed" << endl;
};
```

```
class B {
public:
std::shared_ptr<A> aptr; // circular dependency
B() { std::cout << "B created" << endl; }
~B() { std::cout << "B destroyed" << endl; }
};
int main()
{
std::shared_ptr<A> a = std::make_shared<A>();
std::shared_ptr<B> b = std::make_shared<B>();
a->bptr = b;
b->aptr = a;
return 0; // Destructor will not called. (memory leak)
}
```

* fixed using weak_ptr

```
class B;
class A {
public:
std::weak_ptr<B> bptr; // use weak_ptr to break circular reference
};
class B {
public:
std::shared_ptr<A> aptr;
};
```

✓ Now, the destructor will be called properly.

Q 4) what happens when you move() a unique_ptr?

→ std::move() transfer ownership from one unique_ptr to another.

→ The old unique_ptr becomes nullptr.

e.g:-

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(100);
std::unique_ptr<int> ptr2 = std::move(ptr1); //ownership transfered.
if (!ptr)
{
std::cout << "ptr1 is nullptr after move" << std::endl;
}
```

Note:- ptr1 is now nullptr, and ptr2 owns the memory.

Q5) Does shared_ptr increases memory overhead? why?

yes, shared_ptr has slight memory overhead because:

1) It maintains a control block (for reference counting)

2) Extra space is required to store the reference count.

3) Atomic operations (increment/decrement reference count) add computational overhead.

e.g:- control Block

```
std::shared_ptr<int> p1 = std::make_shared<int>(50);
std::shared_ptr<int> p2 = p1; // Reference count increases
std::cout << "Reference count:" << p1.use_count() << std::endl;
```

Note:- Using std::make_shared<T>() reduces overhead by allocating control block & object together.

summary of smart pointers.

smart pointer	ownership type	copyable?	Reference count?	Use case
unique_ptr	single-ownership	NO	NO	Exclusive ownership a resource
shared-ptr.	shared-ownership	yes	yes	when multiple pointers share ownership
weak_ptr	Non-owning	NO	NO	prevent circular references.

Quick Revision For Interview

1) which smart pointer prevents memory leaks in c++?

→ unique-ptr (auto-delete memory)

2) which smart pointer supports reference counting?

→ shared-ptr.

3) which smart pointer avoids circular dependency?

→ weak-ptr.

4) can we copy a unique-ptr?

→ No, only std::move() is allowed

5) why use std::make_shared<T>()?

→ Reduces memory overhead & improves performance.

Using & namespace keywords

The namespace keyword and using keyword are used in c++ to manage scope & avoid name conflicts.

1) namespace Keyword

The namespace keyword is used to define a scope for identifiers (variable, functions, classes, etc.) to prevent name conflicts.

e.g:-

```
#include <iostream>

namespace myNamespace {
    int value = 42;
    void display() {
    std:: cout <<"Inside myNamespace: value = " <<value << std:: endl;
    }
}
int main()
{
std:: cout << myNamespace :: value << std :: endl;  // 42
myNamespace:: display(); // o/p calls the function insid myNamespace.
return 0;
}
```

Note:- 1) Avoids naming conflicts in large projects

2) Useful when multiple libraries define the same function names.

3) Helps organize codes better

2) Using Keyword

The using keyword simplifies access to namespace members.

```
#include <iostream>
using namespace std; // Now we don't need to write std:: before cout/cin
int main()
{
cout<< "Hello, world!" <<endl;
return 0;
}
```

Downside :

-Imports everything which can cause name conflicts if multiple namespace have the same function/variable name.

* using for specific members (Recommended)

```
#include <iostream>
namespace Alpha {
    void greet() {
    std::cout <<"Hello from Alpha!" <<std::endl;
    }
}
using Alpha::greet; // only imports 'greet()', not the entire namespace
int main()
{
    greet(); // No need for Alpha::greet()
    return 0;
}
```

Note:- safer than using namespace Alpha; because it avoids name conflicts.

* using with Type Aliases

The using keyword can also be used to create type aliases (similar to typedef)

```
#include <iostream>
#include <vector>

// Instead of writing std::vector<int> every time
using IntVector = std::vector<int>;
int main()
{
IntVector numbers = {1, 2, 3, 4};
  for (int num : numbers)
  {
   std::cout << num << " ";
  }
  return 0;
}
```

Note :- More readable than typedef std::vector<int> IntVector;

Summary

Feature	namespace	using
Purpose	Defines a scope to prevent naming conflicts.	simplifies access to namespace members.
Usage	namespace MyNS { int x; }	using MyNS::x; using namespace std;
Full Namespace Access	std::cout	
Type Alias	Not applicable	using MyType = std::vector<int>;

Templates

Templates in c++ allow you to write generic & reusable code. Instead of writing different functions or classes for different data types, you can create a single template that works for all types.

Q1) Function Templates

A function template allows a function to operate on different data types without rewriting the code.

```
#include <iostream>
using namespace std;
template <typename T> // or class T

T add (T a, T b)
  {
    return a+b;
  }
int main()
{
cout << add(5,10) <<endl;
cout << add (5.5, 2.2) <<endl;
return 0;
}
```

Key points :- template <typename T> makes the function generic

T is replace by actual data types during function calls.

The reduces code duplication & increases flexibility.

Q2) class Templates

A class template allows defining generic classes that work with any data type.

```
#include <iostream>
using namespace std;
template <typename T> class Box {
        private:
          T value;
        Public :
        Box (T v) : value (v) { }
        void display() { cout <<"value : "<<value << endl; }
    };

int main()
{
 Box<int> intBox (100);
 intBox.display(); // value : 100
 Box<string> strBox ("Hello");
 strBox.display(); //value : Hello
  return 0;
}
```

Keypoints: 1) The class Box<T> can store any type (int, string, double, etc

2) Box<int> & Box<string> are specializations of the generic class.

3) Useful for container class like Stack, Queue, and vector.

Q3) Template specialization

Sometimes, you want a different implementation for a specific type

```
#include <iostream>
using namespace std;
template <typename T>
class Printer {
 Public:
  void print (T value) {
    cout <<"Generic print : "<< value <<endl;
    }
  };

//Specialization for char*

template < >
class Printer <char*> {
Public:
 void print (char* value) {
 cout <<"Printing a C-style string : "<<value <<endl;
   }
 };
int main()
{
Printer<int> P1;
P1.Print(10); // Generic print : 10
Printer<char*> P2;
P2.print("Hello");

return 0;
}
```

24) Variadic Templates

variadic templates allow you to pass multiple parameters of different types.

```
#include <iostream>
using namespace std;
template <typename T, typename... Args>
void print(T first, Args... rest) {
    cout << first << " ";
  if constexpr (sizeof...(rest) > 0)
    {
     print(rest...);
     }
   }
int main()
 {
 print(1, 2.5, "Hello", 'A');
 return 0;
 }
```

Key points:- i) Args... rest allows any number of arguments

ii) sizeof... (rest) checks the number of remaining arguments.

iii) Used in logging, debugging, and forwarding arguments.

STL (Standard Template Library)

STL is a collection of pre-defined template classes and functions that provide efficient data structures and algorithms. It saves time and improves code efficiency

1) Components of STL

STL is divided into three main components.

Component	Description	Examples
Containers	predefined data structures that store objects.	vector, list, set, map
Algorithms	functions that operate on containers	sort(), binary_search(), find()
Iterators	used to traverse containers	begin(), end(), next(), prev()

2) Containers in STL

STL provides three types of containers

i) Sequence containers (Linear storage)

ii) Associative containers (sorted / unordered storage)

iii) unordered containers (fast access via hashing)

1) Sequence containers (Linear Data structures)

i) vector (Dynamic Array)

- fast random access (O(1) or vec[i]).

ii) - Efficient insertion/deletion at the end (O(1)).

iii) - Not efficient for insertion in the middle (O(n)).

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
vector<int> v = {10, 20, 30};
v.push_back(40); //Insert at the end
v.pop_back(); // Remove last element
for (int x : v) cout << x << " " << endl;
}
```

* common functions:

→ push_back(x), pop_back()

→ size(), empty()

→ begin(), end(), clear()

→ sort(v.begin(), v.end())

2) List (Doubly Linked list)

→ Efficient insertion/deletion anywhere (O(1)).

→ slower than vector for random access.

e.g:-

```
#include <iostream>
#include <list>
using namespace std;
int main() {
list<int> l = {10, 20, 30};
l.push_front(5); //Insert at beginning
l.push_back(40); //Insert at end

for (int x : l)
{
cout << x << " ";

}
}
```

* common functions

→ push_front(x), push_back(x)

→ pop_front(), pop_back()

→ size(), clear(), sort()

3) Deque (Double-ended Queue)

→ fast insertion/deletion at both ends (O(1)).

→ Slower than vector for random access.

e.g:-

```
#include <iostream>
# include <deque>
using namespace std;
int main() {
deque<int> dq = {10, 20, 30};
dq.push_front(5);
dq.push_back(40);

for (int x : dq)
{
cout << x << " ";
}
}
```

* common functions

→ push_front(x), push_back(x)

→ pop_front(), pop_back()

→ size(), clear(), at(index)

. Associative Containers (Key-value storage)

used when we need sorted access or fast searching

) set (sorted unique Elements)

No duplicate values allowed.

Elements are always sorted.

Insertion & search: O(log n) (BST)

e.g:-

```
#include <iostream>
# include <set>
using namespace std;
int main()
{
 set<int> s = {30, 10, 20, 10};
  s.insert(25);
  for (int x : s)
   {
   cout<<x<<" ";
   }
}
```

* common functions

→ insert(x), erase(x), find(x)

→ begin(), end()

→ lower_bound(x), upper_bound(x)

ii) Map (key-value pairs)

→ stores data as {key, value} pairs.

→ keys are always sorted.

→ search, insert, delete : O(log n).

```
#include <iostream>
#include <map>
using namespace std;
int main(){
map<int, string> m;
m[1] = "Apple";
m[2] = "Banana";
m.insert({3,"cherry"});
for (auto it : m)
{
cout << it.first << " -> " << it.second << endl;
}
}
```

O/P

```
1 -> Apple
2 -> Banana
3 -> cherry
```

* common functions

- Insert({key, value}), erase(key), find(key)
- begin(), end()
- count(key), size()

3) unordered Containers (Fastest Lookup)

uses hashing, so searching takes O(1) on average.

i) Unordered set

- faster than set, but not sorted.

```
#include <iostream>
# include <unordered_set>
using namespace std;
int main() {
unordered_set<int> us = {10, 5, 15, 5}; //No duplicate
us.insert(20);
for(int x : us)
{
cout<<x<<" "; //output order is random.
}
}
```

* Common functions:

Same as set, but O(1) average time complexity.

ii) unordered map

- faster than map, but keys are not sorted.

```
#include <iostream>
# include <unordered_set>
using namespace std;
int main() {
unordered_set<int> us = {10, 5, 15, 5}; //No duplicate
us.insert(20);
```

```
    for (int x : us)
    {
     cout << x << " ";
    }
}
```

* common function

Same as set, but O(1) average time complexity.

ii) unordered_map

→ faster than map, but keys are not sorted.

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main()
{
unordered_map<int, string> um;
 um[1] = "Apple";
 um[2] = "Banana";
 um.insert({3, "cherry"});
for (auto it : um)
 {
 cout << it.first << " -> " << it.second << endl;
 }
}
```

* use when speed matters more than sorting

4) STL Algorithms

STL provides built-in functions for common operations

i) Sorting

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
vector <int> v = {3,1,4,2};
sort (v.begin(), v.end()); //Sorting
for (int x : v)
  {
  cout << x << " ";
  }
}
```

ii) Binary Search

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
vector <int> v = {10, 20, 30, 40};
if (binary_search (v.begin(), v.end(), 20))
  {
  cout << "found";
  } else
      cout << "Not found";
}
```

5) STL Iterators

Used to traverse containers

Iterator Type	Use case
begin()	Point to first element
end()	points after last element
rbegin()	Reverse iterator
advance (it, n)	move iterator by n steps.

Summary

i) vectors for dynamic arrays
ii) Sets & maps for unique/sorted storage
iii) unordered sets/maps for fast lookups.
iv) STL algorithm (sort, binary_search)

STL Coding problems

Q1) Remove Duplicates from a vector

i/p → { 4, 2, 2, 8, 3, 3, 1 }

o/p → 1 2 3 4 8

Hint :- Use set to store unique elements.

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;
int main()
{
vector <int> v = {4, 2, 2, 8, 3, 3, 1};
set <int> ~~set~~ s(v.begin(), v.end()); // store unique elements.

 for (int x : s)
   {
   cout << x << " "; // o/p 1 2 3 4 8
   }
}
```

Q2) Find the second largest Element

I/P :- 12, 35, 1, 10, 34, 1

O/P : 34

Hint :- use set to remove duplicates, then get the second last eleme

```
#include <iostream>
#include <set>
using namespace std;
```

```
int main()
{
vector<int> V = {12, 35, 1, 10, 34, 1};
set<int> S(V.begin(), V.end());

auto it = S.rbegin(); // Iterator to the largest element
 ++it; // move to the second largest element.
cout<< *it; // o/p : 34
}
```

Q3) Find frequency of Elements

i/p :- 10, 20, 10, 30, 20, 10

o/p :- 10 -> 3
20 -> 2
30 -> 1

Hint :- Use map<int, int> to store frequency

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;
int main(){
vector<int> V= {10, 20, 10, 30, 20, 10};
map<int, int> freq;
for (int x : V)
 {
  freq[x]++; //count occurrences
 }
for (auto it : freq)
 cout<< it.first <<" -> " << it.second <<endl;
```

Q4) sort a map by value

I/P :- Alice -> 85, Bob -> 90, charlie -> 78

o/P :- Bob -> 90

Alice -> 85

charlie -> 78

Hint: Use vector<pair<int, string>> to sort by value.

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
map<string, int> scores = {{"Alice", 85}, {"Bob", 90}, {"charlie", 78
vector<pair<int, string>> sortedScores;

for (auto & it : scores)
sortedscores.push_back({it.second, it.first}); //swap key-value
sort(sortedscores.rbegin(), sortedScores.rend()); // sord descen

for (auto it : sorted scores)
cout << it.second << "->" << it.first << endl;

}
```

Q5) Find the Kth smallest Element.

I/P :- 7, 10, 4, 3, 20, 15 , K = 3

output :- 7

Hint :- Sort & pick the k-1 index

Soln :-

```
#include <iostream>
#include <vector>
# include <algorithm>
using namespace std;
int main()
{
vector <int> V= {7, 10, 4, 3, 20, 15};
  int K=3;
  sort (V.begin(), V.end()); // sort in ascending order
  Cout << V[K-1]; //O/P :- 7
}
```

Q6) Find the most frequent Element

I/P :- 1, 2, 3, 2, 3, 3, 4, 4, 4, 4

O/P : 4

```
# include<iostream>
# include <vector>
# include <map>
using namespace std;
int main() {
vector<int> V = {1, 2, 3, 2, 3, 3, 4, 4, 4, 4};
map<int, int> freq;
int maxfreq = 0, maxElement = -1;
```

```
for (int x : v)
{
 freq[x]++;
if (freq[x] > maxfreq) {
    maxfreq = freq[x];
    maxElement = x;
    }
}
cout <<" most frequent Element :" <<maxElement;
}
```

Q7) Reverse a string using stack

e.g:-

i/p : Hello

o/p : olleH

```
#include <iostream>
# include < stack>
using namespace std;
int main() {
 string str = "HELLO";
 stack<char> s;
 for (char c : str) {
    s.push(c);
  }
 while ( !s.empty()) {
    cout <<s.top();
    s.pop();
  }
}
```

Q8) merge Two sorted Vectors

I/P:- {1, 3, 5}, {2, 4, 6}

o/P:- {1, 2, 3, 4, 5, 6}

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
vector<int> a = {1, 3, 5};
vector<int> b = {2, 4, 6};
vector<int> result (a.size() + b.size());
merge (a.begin(), a.end(), b.begin, b.end()), result.begin());
for (int x : result)
  {
  cout <<x<<" ";
  }
}
```